

## Capitolo 3

# Stack di Attivazione

Durante l'esecuzione di un programma, uno stack contiene i *frame* (o *record di attivazione*) delle procedure. Ogni frame contiene i parametri di ingresso, le variabili locali e i temporanei della procedura a cui fa riferimento. Inoltre i frame sono legati fra di loro in una *catena statica* che permette a una procedura di accedere alle variabili locali dell'ultima istanziazione della procedura che la include sintatticamente nel testo del programma.

Si consideri per esempio il programma in Figura 3.1. La Figura 3.2.(a) mostra lo stack di attivazione al momento dell'inizio dell'esecuzione della funzione `prettyprint`. Lo stack contiene il parametro di ingresso `tree`, un link e il frame per `prettyprint`. Tale frame contiene la variabile locale `output`, dei temporanei e lo spazio per dei parametri di uscita, al momento non utilizzati. La Figura 3.2.(b) mostra come lo stack si evolve al momento della chiamata della procedura `show`. Lo spazio non usato del frame di `prettyprint` viene adesso utilizzato per i due parametri attuali di `show`. Inoltre il link della catena statica permette di accedere alle variabili locali dell'ultima attivazione di `prettyprint`. Questo perché `show` è sintatticamente racchiusa dentro `prettyprint` (Figura 3.1). Anche questa volta, il frame di `show` contiene dello spazio per i parametri in uscita, spazio al momento non utilizzato. La Figura 3.2.(c) mostra l'evoluzione dello stack al momento di una delle due chiamate ricorsive di `show` all'interno di `show`. I parametri sono stati scritti nello spazio a essi riservato, e il puntatore di catena statica punta ancora una volta al frame dell'ultima attivazione di `prettyprint`. La Figura 3.3.(d) mostra come lo stack si evolve al momento della chiamata di `indent` all'interno dell'ultima attivazione di `show`. Questa volta il puntatore di catena statica punta al frame dell'ultima istanziazione di `show`, poiché `indent` è sintatticamente racchiusa dentro `show` (Figura 3.1). Infine, la Figura 3.3.(e) mostra l'evoluzione dello stack al momento della chiamata a `write` all'interno di `indent`. Questa volta `write` è sintatticamente racchiuso dentro `prettyprint`, e quindi il puntatore di catena statica di questa attivazione di `write` deve puntare al frame dell'ultima istanziazione di `prettyprint`.

### 3.1 Catena statica e coordinate di accesso

In generale, quale puntatore di catena statica deve essere usato al momento della chiamata di una procedura? Occorre far sì che il puntatore di catena statica che precede il frame di una procedura  $p$  punti al frame dell'ultima istanziazione della procedura  $q$  che sintatticamente racchiude  $p$ . A tal fine, si potrebbe pensare di decorare i frame col nome della procedura a cui appartengono. Al momento della chiamata a  $p$ , si cerca il frame per la procedura  $q$  e si fa puntare a esso il puntatore di catena statica per  $p$ . Questa tecnica ha almeno due svantaggi:

1. è necessario cercare il frame di  $q$  all'interno dello stack; questa ricerca comporta dei confronti fra stringhe, potenzialmente lenti;
2. due procedure potrebbero avere lo stesso nome: è necessario disambiguarle a tempo di compilazione.

Per questo motivo tale tecnica non è mai usata. Al contrario, si usa la seguente regola dell'*antenato-fratello-figlio*. Si supponga che  $q$  chiami  $p$ :

```

type tree = {key: string, left: tree, right: tree}

function prettyprint(tree: tree) : string =
  let
    var output := ""

    function write(s: string) =
      output := concat(output,s)

    function show(n: int, t: tree) =
      let function indent(s: string) =
        (for i := 1 to n do write(""));
        output := concat(output,s);
        write("\n")
      in if t = nil then indent(".")
        else (indent(t.key);
              show(n+1,t.left);
              show(n+1,t.right))
      end
    in show(0,tree); output
  end
end

```

Figura 3.1: Un programma Tiger di esempio.

- se  $p$  è un figlio di  $q$ , allora il puntatore di catena statica per  $p$  punta al frame di  $q$  (che è in cima allo stack). Questo è esemplificato in Figura 3.2.(b), nel momento cioè in cui `prettyprint` chiama il figlio `show`;
- se  $p$  è un fratello di  $q$ , allora il puntatore di catena statica di  $p$  sarà lo stesso di quello di  $q$ . Questo è esemplificato in Figura 3.2.(c), nel momento cioè in cui `show` chiama ricorsivamente se stesso (`show` è un fratello di `show`);
- se  $p$  è un antenato  $k$  generazioni più vecchio di  $q$ , allora si torna indietro sulla catena statica di  $k$  passi, e si usa il puntatore di catena statica del record ivi trovato come puntatore di catena statica per  $p$ . Questo è esemplificato in Figura 3.3.(e), nel momento cioè in cui `indent` chiama `write`. Dal momento che `write` è un antenato di una generazione più vecchio di `indent`, occorre tornare indietro di un passo sulla catena statica. Finiamo quindi sul frame per la seconda chiamata a `show` (Figura 3.3.(d)). Il suo puntatore di catena statica punta al frame di `prettyprint`, che viene quindi copiato come puntatore di catena statica del frame per `write` (Figura 3.3.(e)).

Come si cerca il valore di una variabile all'interno dello stack? Consideriamo la lettura del valore di `output` alla fine della procedura `prettyprint`. Dal momento che `output` è una variabile locale a `prettyprint`, il suo valore lo troviamo all'interno del frame per `prettyprint`, come mostrato in Figura 3.2.(a). Essendo la prima (e unica) variabile locale di `prettyprint`, sappiamo anche che la troveremo subito sotto l'inizio del frame per `prettyprint`, cioè subito sotto il frame pointer. Consideriamo adesso la lettura del parametro `s` nella procedura `write`. Ancora una volta, la variabile `s` è locale alla procedura `write` ma, questa volta, si tratta di un parametro di input. Concludiamo che il suo valore si trova subito *prima* dell'inizio del frame per `write`, come mostrato in Figura 3.3.(e). Basta saltare lo spazio per il link. Consideriamo infine la lettura della variabile `output` all'interno della procedura `write`. Questa volta stiamo leggendo il valore di una variabile *esterna* a `write`. Tale variabile è definita un livello di scope più all'esterno di `write`. Concludiamo che dobbiamo tornare indietro di un passo sulla catena statica e leggere la prima variabile locale del frame ivi trovato. La Figura 3.3.(e) mostra la correttezza di questo ragionamento.

In conclusione, l'accesso a una variabile richiede la conoscenza di due *coordinate di accesso* ( $k, n$ ):

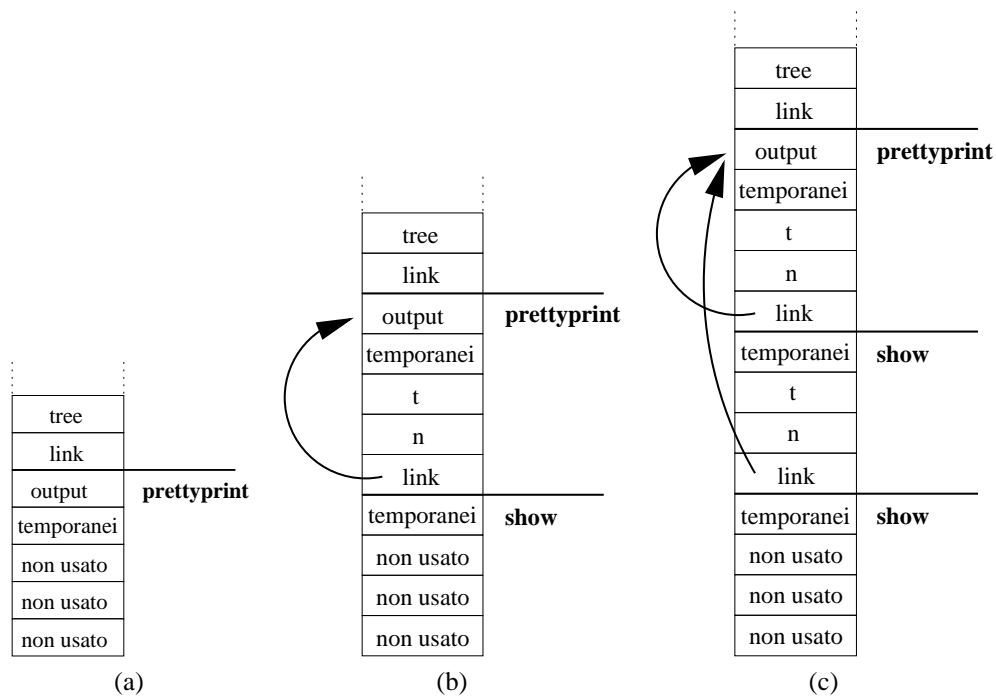


Figura 3.2: Frame per il programma in Figura 3.1.

- $k$  indica di quanti passi tornare indietro sulla catena statica per trovare il frame  $f$  in cui la variabile è contenuta; alternatively,  $k$  può identificare il *livello* (o *scope*) in cui una variabile è definita. L'accesso a tale variabile richiederà di tornare indietro di tanti passi quanta è la differenza fra il livello attuale e quello della variabile;
- $n$  indica di quanto muoversi (in più o in meno) dall'inizio di  $f$  per trovare il valore della variabile.

Entrambe queste coordinate sono **note a tempo di compilazione**:  $k$  è determinabile, in un linguaggio a scope statico, calcolando la differenza fra il livello dell'uso di una variabile e il livello della sua definizione;  $n$  è determinabile contando quante variabili locali o parametri sono presenti nella procedura che definisce la variabile, e conoscendo la dimensione di ogni singola variabile.

Si noti in particolare che non serve memorizzare il nome della variabile sullo stack. Questa considerazione non sarebbe più vera in un linguaggio a scope dinamico.

## 3.2 Frame e livelli per la compilazione di Tiger

Come abbiamo visto nella sezione precedente, il codice generato per un programma Tiger dovrà effettuare le seguenti operazioni:

- calcolare il puntatore di catena statica adeguato a una chiamata di procedura; tale puntatore verrà passato alla procedura come il primo dei suoi parametri;
- accedere al valore di una variabile tramite una sequenza di indirezioni sulla catena statica e uno spostamento finale. La lunghezza di questa sequenza e lo spostamento finale dipendono dalle coordinate di accesso della variabile.

Per permettere la generazione di tale codice, usiamo delle strutture dati ausiliarie che, in fase di compilazione, descrivono la struttura che i frame avranno a tempo di esecuzione. Mentre un oggetto di tipo `Frame/Frame.java` tiene conto dei dettagli implementativi di un frame su una specifica architettura,

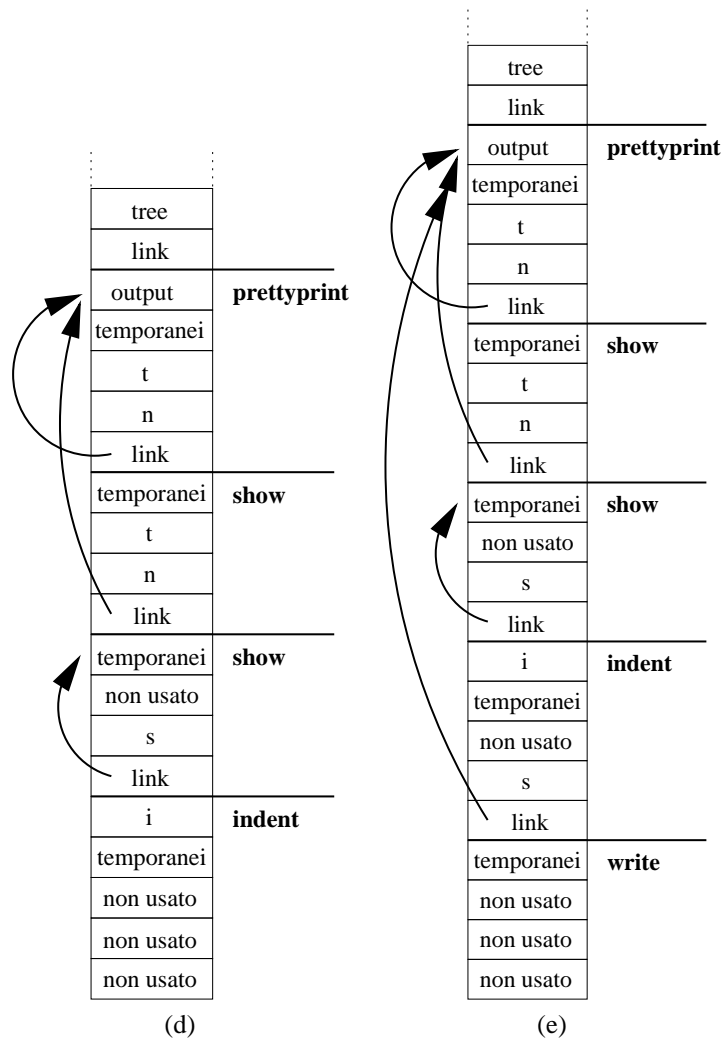


Figura 3.3: Frame per il programma in Figura 3.1.

un oggetto di tipo `Translate/Level.java` descrive, a più alto livello, la relazione di parentela fra i frame delle varie procedure del programma.

Iniziamo col mostrare una classe i cui oggetti implementano *temporanei*, cioè variabili capaci di contenere un valore. Questi temporanei possono essere visti come i registri di una macchina astratta a infiniti registri.

```
Temp/Temp.java
```

```
package Temp;
```

```
public class Temp {
    private static int count = 0;
    private int num;

    public String toString() { return "t" + num; }

    public Temp() { num = count++; }
}
```

```
public Temp(int n) { num = n; }
}
```

Useremo una classe `Temp/Label.java` che descrive delle *etichette*, cioè dei punti di programma etichettati con un nome.

```
Temp/Label.java
```

```
package Temp;
import Symbol.Symbol;
```

```
public class Label {
```

---

Il nome del punto di programma e un contatore usato nel caso in cui si vogliano dei nomi arbitrari ma distinti per `Label` create in successione

---

```
private String name;
private static int count;

public String toString() {return name;}
```

---

Vari costruttori

---

```
public Label(String n) { name = n; }

public Label() { this("L" + count++); }

public Label(Symbol s) {
    this(s.toString() + "_" + count++);
}
}
```

La classe astratta `Frame/Access.java` descrive una zona di memoria che permette di accedere a un valore.

```
Frame/Access.java
```

```
package Frame;
import Temp.Temp;
import Temp.Label;
```

```
public abstract class Access {
```

---

Questo metodo genera il codice che permette di accedere al valore a partire dal codice, fornito come parametro, che permette di accedere al frame pointer

---

```
public abstract Tree.Exp exp(Tree.Exp base);
}
```

Una lista di `Frame/Access.java` è implementata dalla classe astratta `Frame/AccessList.java`.

La classe astratta `Frame/Frame.java` descrive il frame di una procedura. È possibile creare un frame specificando il numero dei parametri della procedura e indicando per ognuno di essi se sfugge dalla procedura (cioè, se è usato da procedure annidate). Inoltre si possono aggiungere variabili locali. Sarà nostra cura istanziare questa classe con sottoclassi concrete per una specifica architettura.

```
Frame/Frame.java
```

```
package Frame;
```

```
public abstract class Frame {
```

---

Un frame conosce il nome della procedura a cui fa riferimento e la lista dei suoi parametri formali

---

```
public Temp.Label name;
public AccessList formals;
```

Dal momento che chi crea i frame non deve conoscere l'architettura per la quale tale frame sono stati pensati, usiamo questo pseudo-costruttore. La lista di booleani specifica se i parametri sfuggono o meno

---

```
abstract public Frame newFrame(Temp.Label name,
                               Util.BoolList formals);
```

Questa procedura permette di aggiungere una variabile locale al frame. Il parametro indica se la variabile sfugge o meno

---

```
abstract public Access allocLocal(boolean escape);
```

Il frame pointer, cioè il registro in cui tale puntatore è memorizzato

---

```
abstract public Temp.Temp FP();
```

Il registro che è usato per contenere il valore di ritorno della funzione il cui metodo è descritto da questo `Frame/Frame.java`

---

```
abstract public Temp.Temp RV();
```

Questo metodo genera del codice assembler che permette di dichiarare una zona di memoria etichettata come `lab` che contiene una stringa `lit`. Il risultato è una stringa, ovvero un pezzo di codice assembler. Dal momento che questo codice dipende dalla specifica architettura su cui si compila, questo metodo è inserito dentro `Frame/Frame.java`

---

```
abstract public String string(Temp.Label lab, String lit);
```

Questi metodi *avvolgono* il codice del corpo di una funzione in modo che esso si interfacci col sistema di ritorno da procedura usato sulla specifica architettura su cui si compila. Abbiamo due metodi perché alcune funzioni ritornano un valore, altre no

---

```
abstract public Tree.Stm procEntryExit1(Tree.Stm body);
abstract public Tree.Stm procEntryExit1(Tree.Exp body);
```

La dimensione di una parola di memoria

---

```
abstract public int wordSize();
}
```

Adesso istanziamo le classi astratte viste sopra per l'architettura Intel. Essenzialmente, tutte le caselle dello stack occupano quattro byte, i parametri di una procedura vengono memorizzati sopra il frame pointer e le variabili locali sotto il frame pointer.

Cominciamo con l'istanziamento della nozione di strumento di accesso a un valore. Dal momento che un valore può essere memorizzato sia sullo stack che in un registro, distinguiamo i due casi con due diverse classi. La classe `Intel/InFrame.java` implementa un valore che viene memorizzato sullo stack. Dal momento che la nozione di link statico non è considerata dalle classi del package `Frame` (verrà recuperata nella classe `Translate/Level.java`), un `Intel/Access.java` contiene solo la coordinata  $n$  di accesso alla variabile contenuta nella casella, cioè solo lo spostamento dal frame pointer.

Intel/InFrame.java

```
package Intel;
import Temp.Temp;
import Temp.Label;

class InFrame extends Frame.Access {
```

---

Questo è lo spostamento dal frame pointer, in numero di byte

---

```
private int offset;

public InFrame(int o) { offset=o; }
```

---

Se base è il codice che ci permette di calcolare il valore del frame pointer, aggiungendo lo spostamento `offset` otteniamo il codice che ci permette di accedere al valore presente sullo stack in tale posizione

---

```
public Tree.Exp exp(Tree.Exp base) {
    return new Tree.MEM(new Tree.BINOP(Tree.BINOP.PLUS,base,
                                      new Tree.CONST(offset)));
}
}
```

Un valore può essere anche memorizzato dentro un registro, identificato da un temporaneo. Abbiamo quindi un'altra sottoclasse di `Frame/Access.java`:

Intel/InReg.java

```
package Intel;
import Temp.Temp;
import Temp.Label;

class InReg extends Frame.Access {
    private Temp temp;

    public InReg() { temp = new Temp(); }
```

---

Per accedere al valore di un registro non serve passare attraverso il frame pointer

---

```
public Tree.Exp exp(Tree.Exp base) { return new Tree.TEMP(temp); }
}
```

La classe `Intel/Frame.java` istanzia `Frame/Frame.java` implementando i suoi metodi in modo consono all'architettura Intel.

Intel/Frame.java

```
package Intel;
import Temp.Temp;
import Temp.Label;

public class IntelFrame extends Frame.Frame {
```

---

Il numero di parametri e di variabili locali nel frame (in multipli di 4)

---

```
private int parCount;
private int localCount;
```

---

I registri usati per contenere il frame pointer e il valore di ritorno della funzione il cui frame è descritto da questo `Intel/IntelFrame.java`

---

```
private Temp FP;
private Temp RV;

public IntelFrame() { this(new Label()); }

public IntelFrame(Label n) {
```

```

name = n;
localCount = 0;
parCount = 0;
FP = new Temp();
RV = new Temp();
}

```

---

Lo pseudo-costruttore crea un altro `Intel/IntelFrame.java` e aggiunge tante strutture di accesso ai parametri formali quanti sono i parametri della procedura. Se un parametro sfugge si usa un registro, altrimenti si alloca spazio sul frame

---

```

public Frame.Frame newFrame(Label n,Util.BoolList e) {
    Intel.IntelFrame result = new IntelFrame(n);

    Frame.AccessList f = result.formals = null;
    Frame.Access a;

    for (; e!=null; e = e.tail) {
        if (e.head) {
            a = new InFrame(result.parCount);
            result.parCount += 4;
        }
        else a = new InReg();

        if (result.formals == null)
            result.formals = f = new IntelAccessList(a,null);
        else {
            f.tail = new IntelAccessList(a,null);
            f = f.tail;
        }
    }

    return result;
}

```

---

Le variabili locali si allocano sotto il frame pointer. Le variabili che sfuggono devono essere allocate sullo stack, le altre possono essere allocate nei registri

---

```

public Frame.Access allocLocal(boolean escape) {
    if (escape) {
        localCount -= 4;
        return new InFrame(localCount);
    }
    else return new InReg();
}

public Temp FP() { return FP; }

public Temp RV() { return RV; }

```

---

Il codice assembler che permette di dichiarare una stringa accessibile tramite un'etichetta

---

```

public String string(Label label, String value) {
    return label + ": .ascii \"" + value + "\"\n";
}

```

---

Se una funzione non ritorna alcun valore, non dobbiamo far nulla al momento del ritorno dalla chiamata. Altrimenti dobbiamo copiare il suo valore di ritorno nel registro usato a tal fine

---

```

public Tree.Stm procEntryExit1(Tree.Stm body) { return body; }

public Tree.Stm procEntryExit1(Tree.Exp body) {
    return procEntryExit1(new Tree.MOVE(new Tree.TEMP(RV),body));
}

public int wordSize() { return 4; }
}

```

Adesso aggiungiamo la seconda dimensione alle coordinate di accesso delle variabili. Specifichiamo cioè in quale *livello* si trova una variabile. Un *livello* è una rappresentazione, a tempo di compilazione, del frame per una procedura a tempo di esecuzione.

```
Translate/Access.java
```

```

package Translate;

public class Access {
    Level home;
    Frame.Access acc;

    public Access(Level h, Frame.Access a) { home = h; acc = a; }
}

```

```
Translate/Level.java
```

```

package Translate;

public class Level {

```

---

Il livello per una procedura contiene il frame per la procedura e un puntatore al livello della procedura che lo include

---

```

    Level parent;
    protected Frame.Frame frame;

```

---

Le coordinate di accesso ai parametri formali della procedura. La coordinata  $n$  è il `Frame/Access.java` contenuta fra i `formals` di `frame`

---

```

    public AccessList formals;

    public Level(Level p, Temp.Label name, Util.BoolList fmls) {
        parent = p;
        frame = parent.frame.newFrame(name, new Util.BoolList(true, fmls));
        formals = null;
        AccessList l = null;

```

---

Dal punto di vista di un livello, il primo parametro di una procedura è il puntatore di catena statica e quindi non serve avere coordinate di accesso per esso, poiché non è usato esplicitamente nel testo del programma. Quindi qui lo scartiamo. Al contrario, il frame non è al corrente della natura di questo parametro, e lo tratterà come un normalissimo parametro per la procedura

---

```

        Frame.AccessList f = frame.formals.tail;

        for (; f != null; f = f.tail)
            if (formals == null)

```

---

Le coordinate di accesso dei parametri avranno questo livello come coordinata  $k$

---

```

        formals = l = new AccessList(new Access(this, f.head), null);
    else {
        l.tail = new AccessList(new Access(this, f.head), null);
        l = l.tail;
    }
}

public Level(Frame.Frame f) {
    parent = null;
    frame = f;
    formals = null;
}

```

---

Le coordinate di accesso delle variabili locali avranno questo livello come coordinata  $k$

---

```

public Access allocLocal(boolean escape) {
    return new Access(this, frame.allocLocal(escape));
}
}

```

Come preliminare alla generazione del codice, vogliamo associare ad ogni funzione un oggetto della classe `Translate/Level.java` che ne descrive il frame. Inoltre, vogliamo associare a ogni uso di variabile le sue coordinate di accesso, cioè un oggetto della classe `Translate/Access.java`. Il Capitolo 4 descrive l'analisi semantica dei programmi Tiger, e mostra come creare questi oggetti durante tale analisi.

### 3.3 Il calcolo delle annotazioni di fuga

Abbiamo visto che la decisione di allocare una variabile locale o un parametro sullo stack o in un registro dipende dal fatto che tale variabile o parametro *sfugga* o meno dallo scope che lo definisce. Occorre cioè sapere se esso è utilizzato in funzioni locali allo scope nel quale è definito.

Le annotazioni di fuga vengono scritte nell'apposito campo `escape` di una `Absyn/VarDec.java` (per le variabili locali) o di una `Absyn/FieldList.java` (per i parametri di una funzione). Si definisce una funzione ricorsiva sulla sintassi astratta del programma Tiger, che viene richiamata dal costruttore della classe `FindEscape/FindEscape.java`. Tale discesa ricorsiva utilizza un ambiente che mappa gli identificatori di variabile in oggetti della seguente classe:

FindEscape/Escape.java

```

package FindEscape;

abstract class Escape {

```

---

Questa è la profondità di scope a cui tale variabile è definita. Quando, durante la discesa ricorsiva, incontriamo una variabile, controlliamo se siamo a una profondità maggiore di quella a cui la variabile è definita. In tal caso, la variabile sfugge, e chiamiamo il metodo `setEscape()`

---

```

    int depth;
    abstract void setEscape();
}

```

Tale classe ha due sottoclassi per variabili locali e parametri formali di una funzione, rispettivamente:

FindEscape/VarEscape.java

```
package FindEscape;

class VarEscape extends Escape {
    Absyn.VarDec vd;

    VarEscape(int d, Absyn.VarDec v) { depth=d; vd=v; vd.escape=false; }
    void setEscape() { vd.escape=true; }
}
```

FindEscape/FormalEscape.java

```
package FindEscape;

class FormalEscape extends Escape {
    Absyn.FieldList fl;

    FormalEscape(int d, Absyn.FieldList f) {
        depth=d; fl=f; fl.escape=false;
    }

    void setEscape() { fl.escape=true; }
}
```

La realizzazione della discesa ricorsiva e la verifica della sua funzionalità fanno parte del progetto di quest'anno.

## Capitolo 5

# Generazione del Codice Intermedio

### 5.1 Modalità di compilazione

Un pezzo di codice Tiger è compilato in un oggetto della classe `Tree/Exp.java` o `Tree/Stm.java`. Ci sono però *modalità* di compilazione diverse, sulla base dell'*uso* che intendiamo fare del codice. Possiamo cioè compilarlo in una modalità `Ex`, nel caso in cui ci interessa avere un valore ritornato dal codice, oppure in modalità `Nx`, nel caso in cui tale valore non ci interessi, e infine in modalità `Cx`, nel caso in cui vogliamo saltare a due etichette specificate, sulla base per esempio del risultato di un test. Questo conduce alla seguente classe, che è un wrapper generico di codice compilato, e permette di trasformarlo in un codice adatto a una delle modalità appena viste.

```
Translate/Exp.java
```

```
package Translate;

public abstract class Exp {
    public abstract Tree.Exp unEx();
    public abstract Tree.Stm unNx();
    public abstract Tree.Stm unCx(Temp.Label t, Temp.Label f);
}
```

Potremmo accontentarci della sola classe `Translate/Exp.java`, ma spesso il contesto in cui un codice si trova ci fa preferire una modalità di compilazione piuttosto che un'altra. Per esempio, in `if e then s1 else s2`, la compilazione di `e` deve essere effettuata in modalità `Cx`. Potremmo compilare `e` in una modalità di default (per esempio, la `Ex`), ottenere del codice *avvolto* nel wrapper `Translate/Exp.java` e quindi applicare il metodo `unCx` con due etichette opportune. Questo però ha l'effetto di aggiungere un test di uguaglianza a 0 del valore di `e`, mentre una semplice istruzione `CJUMP` sarebbe stata sufficiente. A fini di ottimizzazione del codice generato, usiamo tre sottoclassi di `Translate/Exp.java` che assumono che il codice che esse avvolgono si comporti di per sé secondo una delle tre modalità viste sopra:

```
Translate/Ex.java
```

```
package Translate;

class Ex extends Exp {
```

---

Questo è il codice *avvolto* da questo wrapper. Come si vede, assumiamo che esso sia del codice che ritorna un valore

---

```
    Tree.Exp exp;

    public Ex(Tree.Exp e) { exp=e; }
```

---

Grazie all'ipotesi sul tipo di codice *avvolto* da questo wrapper, questa modalità di compilazione è immediata e non aggiunge codice inutile

---

```
public Tree.Exp unEx() { return exp; }
```

---

Se il valore di ritorno non ci serve, lo scartiamo

---

```
public Tree.Stm unNx() { return new Tree.EXP(exp); }
```

---

Se vogliamo trasformarlo in un test, aggiungiamo un controllo di uguaglianza a zero. Prima però controlliamo il caso speciale in cui l'espressione avvolta è una costante. In tal caso (abbastanza frequente) possiamo generare del codice più efficiente che salta a una delle due etichette sulla base del valore della costante, senza bisogno di controllarlo a tempo di esecuzione

---

```
public Tree.Stm unCx(Temp.Label t, Temp.Label f) {
    if (exp instanceof Tree.CONST)
        return ((Tree.CONST)exp).value==0 ?
            new Tree.JUMP(f) : new Tree.JUMP(t);

    return new Tree.CJUMP(Tree.CJUMP.EQ, exp, new Tree.CONST(0), f, t);
}
}
```

Translate/Nx.java

```
package Translate;
```

```
class Nx extends Exp {
```

---

Questa volta il codice non ritorna nulla, quindi è un oggetto della classe `Tree/Stm.java`

---

```
Tree.Stm stm;
```

```
public Nx(Tree.Stm s) { stm=s; }
```

---

Non è possibile che un codice che non ritorna un valore venga compilato in un contesto in cui serve un valore di ritorno

---

```
public Tree.Exp unEx() { throw new Error("unEx"); }
public Tree.Stm unNx() { return stm; }
```

---

Non è possibile che un codice che non ritorna un valore venga compilato in un contesto in cui serve effettuare un test sulla base di tale valore

---

```
public Tree.Stm unCx(Temp.Label t, Temp.Label f) {
    throw new Error("unCx");
}
}
```

Translate/Cx.java

```
package Translate;
```

---

Questa classe è ancora astratta! Vedremo dopo un'istanziamento

---

```
abstract class Cx extends Exp {
```

Per trasformare un test in un'espressione che ritorna un valore, usiamo un registro ausiliario  $r$  e compiliamo il tutto come segue:

```

    r = 1
    unCx(t, f)
    f : r = 0
    t :
    ritorna r

```

---

```

public Tree.Exp unEx() {
    Temp.Temp r = new Temp.Temp();
    Temp.Label t = new Temp.Label();
    Temp.Label f = new Temp.Label();

    return new Tree.ESEQ(
        new Tree.SEQ(new Tree.MOVE(new Tree.TEMP(r), new Tree.CONST(1)),
            new Tree.SEQ(unCx(t, f),
                new Tree.SEQ(new Tree.LABEL(f),
                    new Tree.SEQ(new Tree.MOVE(new Tree.TEMP(r), new Tree.CONST(0)),
                        new Tree.LABEL(t))))),
        new Tree.TEMP(r));
    }

```

Per trasformare un test in un codice che non ritorna nulla, compiliamo il codice come  $unCx(t, t); t :$

---

```

public Tree.Stm unNx() {
    Temp.Label t=new Temp.Label();

    return new Tree.SEQ(unCx(t, t), new Tree.LABEL(t));
}

```

La compilazione in modalità Cx dipende dal tipo di test e deve essere implementata dalle sottoclassi (vedi Translate/RelCx.java)

---

```

public abstract Tree.Stm unCx(Temp.Label t, Temp.Label f);
}

```

Un esempio di sottoclasse di Translate/Cx.java è la seguente, che serve a compilare i test di confronto:

Translate/RelCx.java

```
package Translate;
```

```
public class RelCx extends Cx {
```

---

Specifichiamo l'operatore di confronto e il codice che contiene la compilazione dei due operandi

```

    private int oper;
    private Exp left, right;

    public RelCx(int o, Exp l, Exp r) { oper=o; left=l; right=r; }

```

La compilazione in modalità Cx richiede di compilare i due operandi in modalità Ex (perché devono ritornare un valore) e di generare un salto condizionato alle due etichette specificate

---

```

    public Tree.Stm unCx(Temp.Label t, Temp.Label f) {
        return new Tree.CJUMP(oper, left.unEx(), right.unEx(), t, f);
    }
}

```

## 5.2 Frammenti di compilazione

La compilazione di un pezzo di codice Tiger ha come effetto la generazione di una serie di *frammenti*. Alcuni saranno frammenti di codice intermedio, altri frammenti per le stringhe. Avremo quindi una classe astratta `Translate/Frag.java` e due sottoclassi concrete:

```
Translate/Frag.java
```

```
package Translate;

abstract public class Frag {
    public Frag next;

    abstract public void print();
}
```

```
Translate/ProcFrag.java
```

```
package Translate;

public class ProcFrag extends Frag {


---




---


    public Tree.Stm body;
    public Frame.Frame frame;

    public ProcFrag(Tree.Stm b, Frame.Frame f, Frag n) {
        body = b; frame = f; next = n;
    }

    public void print() {
        System.out.println("code fragment (" + frame.name + "):");

```

---

Stampiamo il codice contenuto nel frammento

---

```
    (new Tree.Print(System.out)).prStm(body);
}
```

```
Translate/DataFrag.java
```

```
package Translate;

public class DataFrag extends Frag {


---




---


    public Temp.Label label;
    public String data;

    public DataFrag(Temp.Label l, String d, Frag n) {
        label = l; data = d; next = n;
    }

    public void print() {
        System.out.println("data fragment (" + label + "):");
        System.out.println(data);
    }
}
```

## 5.3 La generazione del codice

La generazione del codice Tiger nel linguaggio intermedio viene effettuata da una classe `Translate/Translate.java` che contiene un metodo per ogni classe di semantica astratta. Per compilare le istanze di una classe astratta *A*, occorre avere già compilato tutte le classi astratte che figurano nella definizione di *A*.

Mentre effettua la compilazione, la classe `Translate/Translate.java` accumula frammenti di codice in una lista `frags`. Alla fine della compilazione verrà richiamato un metodo che stampa tutti i frammenti accumulati.

```
Translate/Translate.java
```

```
package Translate;
```

```
public class Translate {
```

---

Queste due etichette dovranno, in fase di assemblaggio, essere linkate a delle routine che stampano un errore e che allocano della memoria, rispettivamente

---

```
    private Temp.Label error = new Temp.Label("error");
    private Temp.Label malloc = new Temp.Label("malloc");
```

---

Questa è la lista di frammenti di compilazione che viene generata durante la compilazione di pezzi di programmi Tiger. È possibile leggerla tramite il metodo `getResult()` e stamparla tramite il metodo `printFrag()`

---

```
    private Frag frags = null;

    public Frag getResult() { return frags; }

    public void printFrag() {
        Frag cursor;

        for (cursor = frags; cursor != null; cursor = cursor.next) {
            cursor.print();
            System.out.println();
        }
    }
}
```

---

Questo metodo stampa il codice compilato `code`. Chiama il metodo di stampa opportuno sulla base del fatto che il codice ritorni un valore o no

---

```
    public static void print(Exp code) {
        if (code instanceof Nx)
            (new Tree.Print(System.out)).prStm(code.unNx());
        else
            (new Tree.Print(System.out)).prExp(code.unEx());
    }
}
```

## 5.4 La traduzione delle dichiarazioni

Le dichiarazioni hanno un significato quasi esclusivamente per il controllo dei tipi effettuato nel Capitolo 4. Esse quindi generano del codice che non effettua alcuna operazione, tranne per la dichiarazione di variabile che genera del codice per inizializzare la variabile.

Aggiungiamo quindi i seguenti metodi a `Translate/Translate.java`.

---

Un metodo che restituisce del codice che non fa e non ritorna nulla

---

```
public Exp Nop() {
    return new Nx(new Tree.EXP(new Tree.CONST(0)));
}
```

Una dichiarazione di funzione genera un codice vuoto. Però deve registrare il codice della funzione fra i frammenti di codice che saranno il risultato della compilazione. Questo è effettuato dal metodo `procEntryExit()`

```
public Exp FunctionDec(Temp.Label name, Exp body, Level l) {
    procEntryExit(l,body);

    return Nop();
}
```

Questo metodo registra il codice di una funzione fra i frammenti generati dal processo di compilazione. Prima di questa registrazione, si preoccupa di chiamare un metodo specifico dell'architettura su cui si compila, che tipicamente espande il codice in modo da copiare il risultato del corpo della funzione su un registro usato come valore di ritorno (si veda `Intel/IntelFrame.java`)

```
public void procEntryExit(Level level, Exp body) {
    Tree.Stm wrappedBody;

    if (body instanceof Ex)
        wrappedBody = level.frame.procEntryExit1(body.unEx());
    else
        wrappedBody = level.frame.procEntryExit1(body.unNx());

    frags = new ProcFrag(wrappedBody,level.frame,frags);
}
```

La dichiarazione di una variabile si compila in del codice che copia il valore di inizializzazione per la variabile in una cella di memoria ottenuta a partire dal frame pointer aggiungendo lo spostamento a cui si trova la variabile (si veda `Frame/Access.java`)

```
public Exp VarDec(Access v, Exp init, Level l) {
    return new Nx(new Tree.MOVE(v.acc.exp(new Tree.TEMP(l.frame.FP())),
                                init.unEx()));
}
```

La dichiarazione di un tipo non ha effetti a livello di compilazione

```
public Exp TypeDec() { return Nop(); }
```

Vediamo alcuni esempi di compilazione. La dichiarazione

```
var a:= 13
```

viene compilata in

```
MOVE(
    MEM(
        BINOP(PLUS,TEMP t0,CONST -4),
        CONST 13)
```

Si noti che `t0` è il registro usato per contenere il frame pointer del programma, e spostandosi in basso di 4 byte si punta alla prima variabile locale del record di attivazione, cioè `a`. Lo stesso codice viene generato per la dichiarazione

```
var a:int := 13
```

Invece la dichiarazione

```
function a():int = 13
```

è compilata in un codice vuoto:

```
EXP(CONST 0)
```

ma ha il side-effect di aggiungere un `Translate/ProcFrag.java` ai frammenti generati per il programma, il quale contiene il codice

```
MOVE(TEMP t3,CONST 13)
```

Si noti che `t3` è il registro usato per ritornare il risultato della funzione, e che questa istruzione `MOVE` è stata generata dal metodo `procEntryExit1()` di `Intel/IntelFrame.java`.

## 5.5 La traduzione dei leftvalue

I leftvalue sono i contenitori dell'informazione che può essere letta e scritta durante l'esecuzione del programma. È quindi naturale che questo sia il caso in cui si usa più pesantemente l'accesso allo stack di attivazione.

Aggiungiamo i seguenti metodi a `Translate/Translate.java`.

---

Il parametro `a` fornisce le coordinate di accesso della variabile a cui vogliamo fare accesso. Il parametro `l` è invece il livello che descrive il record di attivazione della funzione a cui appartiene il codice che stiamo generando. Per accedere alla variabile indicata da `a`, dobbiamo generare del codice che torna indietro sulla catena statica di tanti passi quanta è la differenza fra il livello `l` e il livello in cui è definita la variabile. Poi si aggiunge lo spostamento all'interno del record di attivazione così trovato

---

```
public Exp SimpleVar(Access a, Level l) {
    Level d = a.home;

    Tree.Exp base = new Tree.TEMP(l.frame.FP());
```

---

Il primo dei parametri formali del record di attivazione contiene il puntatore di catena statica. Generiamo del codice che legge il suo valore (si veda `Frame/Access.java`)

---

```
    for (; l != d; l = l.parent) base = (l.frame.formals.head).exp(base);
```

---

Alla fine generiamo del codice che aggiunge lo spostamento all'interno del record di attivazione in cui è contenuta la variabile

---

```
    return new Ex(a.acc.exp(base));
}
```

---

Se vogliamo accedere al campo di un record `e.f`, dobbiamo fornire il codice `record` che permette di accedere al valore di `e`. Questo codice deve essere compilato in modalità `Ex` poiché deve restituirci un puntatore in memoria a cui sommiamo lo spostamento `offset` che ci permette di raggiungere il campo `f`. Se supponiamo che tutti i campi occupino lo stesso spazio di memoria, questo significa sommare `offset`, moltiplicato per la dimensione di una parola, al valore di `e`

---

```
public Exp FieldVar(Exp record, int offset, Level l) {
    offset *= l.frame.wordSize();

    return new Ex(new Tree.MEM(new Tree.BINOP(Tree.BINOP.PLUS,
        record.unEx(), new Tree.CONST(offset))));
}
```

Se vogliamo accedere a un elemento di un array, dobbiamo compilare in modalità Ex l'espressione che ritorna l'array e l'indice a cui accedere. Quindi si somma alla base dell'array il valore dell'indice moltiplicato per la dimensione di una parola. Si noti che mentre nel caso dell'accesso a un campo questa moltiplicazione veniva fatta a tempo di compilazione, qui la facciamo a tempo di esecuzione, dal momento che il compilatore non conosce il valore dell'indice dell'array a cui si fa accesso

```
public Exp SubscriptVar(Exp array, Exp index, Level l) {
    return new Ex(new Tree.MEM(new Tree.BINOP(Tree.BINOP.PLUS,
        new Tree.MEM(array.unEx()),new Tree.BINOP(Tree.BINOP.MUL,
            index.unEx(),new Tree.CONST(l.frame.wordSize())))));
}
```

Vediamo alcuni esempi di compilazione di leftvalue. Si consideri il programma

```
let
    function a(n:int):int =
        let
            function b(m:int):int = m + n
        in
            2
        end
    in
        1
    end
```

L'espressione `m + n` viene compilata in

```
BINOP(PLUS,
    MEM(
        BINOP(PLUS,TEMP t4,CONST 4)),
    MEM(
        BINOP(PLUS,
            MEM(BINOP(PLUS,TEMP t4,CONST 0)),
            CONST 4)))
```

L'espressione `BINOP(PLUS,TEMP t4,CONST 4)` è il primo parametro della funzione `b`, cioè `m`, mentre l'espressione `BINOP(PLUS,TEMP t4,CONST 0)` è il puntatore di catena statica di tale funzione. Come si vede, il codice dice di leggere il contenuto di tale puntatore e sommare 4 al risultato, in modo da accedere al primo parametro della funzione `a`, cioè `n`. I valori di `m` e di `n` vengono infine sommati. Si noti che l'espressione `BINOP(PLUS,TEMP t4,CONST 0)` potrebbe essere semplificata perché sommare 0 non serve.

Si consideri adesso il programma

```
let
    type list = {head: int, tail:list}
    function a(l:list):int = l.tail.head
in
    1
end
```

L'espressione `l.tail.head` viene compilata in

```
MEM(
    BINOP(PLUS,
        MEM(
```

```

BINOP(PLUS,
  MEM(BINOP(PLUS,TEMP t2,CONST 4)),
  CONST 4)),
CONST 0))

```

L'espressione `BINOP(PLUS,TEMP t2,CONST 4)` punta al parametro `l`, per cui con l'espressione `MEM(BINOP(PLUS,TEMP t2,CONST 4))` accediamo al valore di `l`. Aggiungendo 4 puntiamo al campo `tail` di `l`. Quindi leggiamo il valore della parola di memoria così individuata, il che ci dà il valore di `l.tail`. Sommando 0 (operazione superflua) otteniamo un puntatore a `l.tail.head`, e leggendo la parola di memoria puntata otteniamo il valore di `l.tail.head`.

Si consideri infine il programma

```

let
  type t = array of int
  function a(n:t, i:int):int = n[i]
in
  1
end

```

L'espressione `n[i]` viene compilata in

```

MEM(
  BINOP(PLUS,
    MEM(
      MEM(
        BINOP(PLUS,TEMP t2,CONST 4))),
    BINOP(MUL,
      MEM(
        BINOP(PLUS,TEMP t2,CONST 8))),
    CONST 4)))

```

L'espressione `BINOP(PLUS,TEMP t2,CONST 4)` ci permette di accedere al valore del parametro `n`. L'espressione `BINOP(PLUS,TEMP t2,CONST 8)` ci permette invece di accedere al parametro `i`. Come si può vedere, il codice dice di eseguire la moltiplicazione per 4 del valore di `i` (4 è la dimensione di una parola), e di sommare il risultato al valore di `n`. Quindi di accedere al contenuto della cella di memoria così individuata.

## 5.6 La traduzione delle espressioni

Le espressioni formano la larga maggioranza del codice Tiger. La loro compilazione può ritornare sia del codice compilato in modalità `Ex` che del codice compilato in modalità `Nx`, dal momento che alcune espressioni (come i cicli `while`) non ritornano alcun valore.

Aggiungiamo i seguenti metodi alla classe `Translate/Translate.java`.

---

Una espressione può essere un `leftvalue`. In tal caso la compilazione del `leftvalue` è già la compilazione dell'espressione

---

```
public Exp VarExp(Exp lvalue) { return lvalue; }
```

---

La costante `nil` è compilata in una espressione intera che ritorna 0

---

```
public Exp NilExp() { return new Ex(new Tree.CONST(0)); }
```

---

Una costante intera è compilata in una espressione che ritorna il valore della costante

---

```
public Exp IntExp(int value) { return new Ex(new Tree.CONST(value)); }
```

Una stringa viene compilata in una etichetta *l*. Però aggiungiamo un frammento di compilazione che contiene la dichiarazione di una costante stringa assembler in una posizione etichettata con *l*

```
public Exp StringExp(String value, Level level) {
    Temp.Label l = new Temp.Label();

    frags = new DataFrag(l, level.frame.string(l, value), frags);

    return new Ex(new Tree.NAME(l));
}
```

La compilazione di una chiamata a una funzione di nome *name*, la compilazione dei cui parametri è fornita in *pars*, chiamata dal livello *caller* e definita nel livello *callee*. Il flag *noResult* indica se la funzione ritorna qualcosa o *void*

```
public Exp CallExp(Temp.Label name, ExpList pars,
                  Level callee, Level caller, boolean noResult){
    Tree.Exp FP;
    Tree.ExpList newPars, cursor;
```

Qui implementiamo la regola del figlio-fratello-antenato. Prima però consideriamo il caso speciale in cui la funzione chiamata non ha alcun livello associato. Questo significa che è una funzione predefinita, che non è interna a nessuno scope. In tal caso, il frame pointer è *null*, cioè la costante 0

```
    if (callee == null) FP = new Tree.CONST(0);
    else {
```

Altrimenti generiamo del codice che preleva il puntatore al frame pointer del chiamante. Questo sarà il codice che ci permette di calcolare il puntatore di catena statica del chiamato nel caso più semplice in cui il chiamato è un figlio del chiamante

```
        FP = new Tree.TEMP(caller.frame.FP());

        if (caller != callee.parent)
```

Altrimenti il chiamante e il chiamato potrebbero essere fratelli. In tal caso il puntatore di catena statica del chiamato si ottiene partendo dal frame pointer del chiamante e leggendo il puntatore di catena statica ivi trovato

```
            if (caller.parent == callee.parent)
                FP = (caller.frame.formals.head).exp(FP);
            else {
```

Infine il caso della chiamata a un antenato. Partiamo dal frame pointer del chiamante e generiamo del codice che torna indietro sulla catena statica fino a raggiungere il livello del chiamato. Quindi ritorna il frame pointer ivi trovato

```
                for (; caller != callee; caller = caller.parent)
                    FP = (caller.frame.formals.head).exp(FP);

                FP = (caller.frame.formals.head).exp(FP);
            }
        }
```

I parametri sono una *Translate/ExpList.java*, ovvero una lista di *Translate/Exp.java*. Li trasformiamo in una istanza di *Tree/ExpList.java*, ovvero in una lista di *Tree/Exp.java*. Richiediamo che ogni parametri sia compilato in modalità *Ex*, perché abbiamo bisogno del suo valore. Si noti che aggiungiamo il codice che calcola il puntatore di catena statica come un parametro aggiuntivo della funzione

```

newPars = cursor = new Tree.ExpList(FP,null);
while (pars != null) {
  cursor.tail = new Tree.ExpList(pars.head.unEx(),null);
  cursor = cursor.tail;
  pars = pars.tail;
}

```

---

Se la funzione non ha alcun valore di ritorno generiamo del codice in modalità Nx, altrimenti del codice in modalità Ex

---

```

if (noResult)
  return new Nx(new Tree.EXP(
    new Tree.CALL(new Tree.NAME(name),newPars)));
else
  return new Ex(new Tree.CALL(new Tree.NAME(name),newPars));
}

```

Si consideri per esempio il seguente programma Tiger:

```

let
  var s:string := "ciao"
  function a(p:string):string = p
  function b():string = a(s)
in
  b()
end

```

La dichiarazione di s viene compilata in

```

MOVE(
  MEM(BINOP(PLUS,TEMP t0,CONST -4)),
  NAME L11)

```

ovvero in del codice che copia nella prima variabile locale l'indirizzo L11, che è l'etichetta di una zona di memoria in cui è stata allocata la stringa ciao. Infatti la compilazione di tale stringa ha l'effetto di creare il seguente frammento dati:

```

data fragment (L11):
L11: .ascii "ciao"

```

La funzione b viene invece compilata in

```

MOVE(
  TEMP t5,
  CALL(
    NAME a_12,
    MEM(BINOP(PLUS,TEMP t4,CONST 0)),
    MEM(
      BINOP(PLUS,
        MEM(BINOP(PLUS,TEMP t4,CONST 0)),
        CONST -4))))

```

ovvero in del codice che copia nel registro di ritorno t5 il risultato della chiamata ad a (a\_12 è l'etichetta usata per il codice di a), passando come parametri il puntatore di catena statica del chiamante (perché a e b sono fratelli) e la prima variabile locale del livello in cui b è definita (cioè del corpo del let).

Si consideri invece il programma Tiger

```

let
  var s:string := "ciao"
  function a(p:string):string =
    let
      function b():string = a(p)
    in
      b()
    end
  in
    a(s)
end

```

La funzione b viene compilata in

```

MOVE(
  TEMP t5,
  CALL(
    NAME a_12,
    MEM(
      BINOP(PLUS,
        MEM(BINOP(PLUS,TEMP t4,CONST 0)),
        CONST 0)),
    MEM(
      BINOP(PLUS,
        MEM(BINOP(PLUS,TEMP t4,CONST 0)),
        CONST 4))))

```

Si noti la differenza col caso precedente. In primo luogo, il primo parametro che è passato ad a, cioè il puntatore di catena statica, è ottenuto tornando indietro di un passo sulla catena statica e quindi leggendo il valore ivi trovato. In secondo luogo, il secondo parametro, cioè p, è ottenuto tornando indietro di un passo sulla catena statica e aggiungendo 4 piuttosto che sottraendo 4, dal momento che qui p è un parametro di a, mentre nell'esempio precedente s è una variabile locale del corpo principale del programma.

Continuiamo ad esaminare i metodi che aggiungiamo a `Translate/Translate.java`.

---

Un'operazione binaria viene compilata di default in modalità Ex o Cx

---

```

public Exp OpExp(Exp left, int oper, Exp right) {
  Tree.Exp l = left.unEx(); Tree.Exp r = right.unEx();
  Exp lt = new Ex(l); Exp rt = new Ex(r);

  switch (oper) {
  case Absyn.OpExp.PLUS:
    return new Ex(new Tree.BINOP(Tree.BINOP.PLUS,l,r));
  case Absyn.OpExp.MINUS:
    return new Ex(new Tree.BINOP(Tree.BINOP.MINUS,l,r));
  case Absyn.OpExp.MUL:
    return new Ex(new Tree.BINOP(Tree.BINOP.MUL,l,r));
  case Absyn.OpExp.DIV:
    return new Ex(new Tree.BINOP(Tree.BINOP.DIV,l,r));
  case Absyn.OpExp.EQ: return new RelCx(Tree.CJUMP.EQ,lt,rt);
  case Absyn.OpExp.NE: return new RelCx(Tree.CJUMP.NE,lt,rt);
  case Absyn.OpExp.LT: return new RelCx(Tree.CJUMP.LT,lt,rt);
  case Absyn.OpExp.LE: return new RelCx(Tree.CJUMP.LE,lt,rt);
  case Absyn.OpExp.GT: return new RelCx(Tree.CJUMP.GT,lt,rt);
  case Absyn.OpExp.GE: return new RelCx(Tree.CJUMP.GE,lt,rt);
  default: throw new Error("Translate.OpExp");
  }
}

```

```

}
}

```

---

La creazione di un record viene lasciata agli studenti!

---

```
public Exp RecordExp(ExpList fields) { return new Ex(new Tree.CONST(1)); }
```

---

Un assegnamento viene compilato in una istruzione MOVE fra il codice per il lato sinistro e quello per il lato destro

---

```
public Exp AssignExp(Exp lvalue, Exp rvalue) {
    return new Nx(new Tree.MOVE(lvalue.unEx(), rvalue.unEx()));
}
```

---

Distinguiamo fra il condizionale semplice e quello a due uscite. Nel primo caso creiamo due etichette, compiliamo il test in modalità Cx fornendo le due etichette come punti di salto e compiliamo il corpo in modalità Nx poiché il condizionale semplice in Tiger non ritorna mai un valore

---

```
public Exp IfExp(Exp test, Exp thenclause) {
    Temp.Label t = new Temp.Label(); Temp.Label f = new Temp.Label();

    return new Nx(new Tree.SEQ(
        test.unCx(t, f),
        new Tree.SEQ(
            new Tree.LABEL(t),
            new Tree.SEQ(
                thenclause.unNx(),
                new Tree.LABEL(f)
            )
        )
    ));
}
```

---

Nel caso del condizionale a due uscite, dobbiamo distinguere il caso in cui i due rami ritornano entrambi void (e quindi vanno compilati in modalità Nx e il risultato è in modalità Nx) da quello in cui invece ritornano qualcosa (e quindi vanno compilati in modalità Ex e il risultato è in modalità Ex)

---

```
public Exp IfExp(Exp test, Exp thenclause, Exp elseclause) {
    Temp.Label t = new Temp.Label();
    Temp.Label f = new Temp.Label();
    Temp.Label end = new Temp.Label();

    if (thenclause instanceof Nx)
        return new Nx(new Tree.SEQ(
            test.unCx(t, f),
            new Tree.SEQ(
                new Tree.LABEL(t),
                new Tree.SEQ(
                    thenclause.unNx(),
                    new Tree.JUMP(end),
                    new Tree.LABEL(f),
                    new Tree.SEQ(
                        elseclause.unNx(),
                        new Tree.LABEL(end)
                    )
                )
            )
        ));
    else {
        Tree.TEMP r = new Tree.TEMP(new Temp.Temp());

        return new Ex(new Tree.ESEQ(
            new Tree.SEQ(
                test.unCx(t, f),
                new Tree.SEQ(
                    new Tree.LABEL(t),
                    new Tree.SEQ(
                        new Tree.MOVE(r, thenclause.unEx()),
                        new Tree.JUMP(end),
                        new Tree.LABEL(f),
                        new Tree.MOVE(r, elseclause.unEx()),
                        new Tree.LABEL(end)
                    )
                )
            ), r));
    }
}
```

La compilazione del ciclo `while` ci dà modo di comprendere l'uso dell'etichetta `_break` gestita dalla classe `Semant/Semant.java`. Essa è l'etichetta a cui saltare in caso si esegua un'istruzione `break`. Compiliamo il ciclo `while` in maniera tale da etichettare l'istruzione successiva al ciclo con l'etichetta `_break`. Quindi l'istruzione `break`, che viene compilata in un salto incondizionato all'etichetta `_break`, ci permetterà di uscire dal ciclo (si veda il metodo successivo)

```
public Exp WhileExp(Exp test, Exp body, Temp.Label _break) {
    Temp.Label start = new Temp.Label();
    Temp.Label t = new Temp.Label();

    return new Nx(new Tree.SEQ(    new Tree.LABEL(start),
                                new Tree.SEQ(    test.unCx(t,_break),
                                new Tree.SEQ(    new Tree.LABEL(t),
                                new Tree.SEQ(    body.unNx(),
                                new Tree.SEQ(    new Tree.JUMP(start),
                                new Tree.LABEL(_break))))));
}

```

Un'istruzione `break` viene compilata in un salto incondizionato all'etichetta `_break`

```
public Exp BreakExp(Temp.Label _break) {
    return new Nx(new Tree.JUMP(_break));
}

```

La compilazione del ciclo `for` deve essere scritta dagli studenti!

```
public Exp ForExp(Exp var, Exp hi, Exp body,
                 Access v, Temp.Label _break) { return Nop(); }

```

La compilazione di un'espressione `let` risulta nella compilazione delle dichiarazioni seguita dalla compilazione del corpo

```
public Exp LetExp(Exp decs, Exp body) { return Append(decs,body); }

```

La compilazione di due espressioni in sequenza consiste nella compilazione della prima in modalità `Nx` (il suo valore, ammesso che esista, viene infatti scartato) e della seconda in modalità `Ex` o `Nx` sulla base del fatto che essa ritorni qualcosa o meno

```
public Exp Append(Exp head, Exp tail) {
    if (tail instanceof Nx)
        return new Nx(new Tree.SEQ(head.unNx(),tail.unNx()));
    else
        return new Ex(new Tree.ESEQ(head.unNx(),tail.unEx()));
}

```

La compilazione della creazione di un array

```
public Exp ArrayExp(Exp size, Exp init, Level l) {
    Temp.Temp s = new Temp.Temp(); Temp.Temp i = new Temp.Temp();
    Temp.Temp b = new Temp.Temp(); Temp.Temp bb = new Temp.Temp();
    int k = l.frame.wordSize();

    Temp.Label l1 = new Temp.Label(); Temp.Label l2 = new Temp.Label();
    Temp.Label l3 = new Temp.Label(); Temp.Label l4 = new Temp.Label();
    Temp.Label l5 = new Temp.Label();

    return new Ex(new Tree.ESEQ

```

Salviamo in due temporanei la dimensione dell'array e il suo valore di inizializzazione

```
( new Tree.SEQ( new Tree.MOVE(new Tree.TEMP(s),size.unEx()),
  new Tree.SEQ( new Tree.MOVE(new Tree.TEMP(i),init.unEx()),
```

---

Se la dimensione è negativa diamo errore

---

```
new Tree.SEQ( new Tree.CJUMP(Tree.CJUMP.LT,new Tree.TEMP(s),
  new Tree.CONST(0),error,l1),
new Tree.SEQ( new Tree.LABEL(l1),
```

---

Altrimenti allochiamo size parole di memoria e poniamo nel temporaneo b l'indirizzo a partire dal quale tali parole sono state allocate. Se b è 0 vuol dire che l'allocazione è fallita e dobbiamo dare errore

---

```
new Tree.SEQ( new Tree.MOVE(new Tree.TEMP(b),new Tree.CALL(
  new Tree.NAME(malloc),
  new Tree.ExpList(new Tree.TEMP(s),null))),
new Tree.SEQ( new Tree.CJUMP(Tree.CJUMP.EQ,new Tree.TEMP(b),
  new Tree.CONST(0),error,l2),
new Tree.SEQ( new Tree.LABEL(l2),
```

---

b sarà il risultato dell'espressione. Per non modificarlo, lo copiamo in un registro bb che verrà incrementato a ogni iterazione della dimensione di una parola

---

```
new Tree.SEQ( new Tree.MOVE(new Tree.TEMP(bb),new Tree.TEMP(b)),
new Tree.SEQ( new Tree.LABEL(l3),
```

---

Eseguiamo size iterazioni

---

```
new Tree.SEQ( new Tree.CJUMP(Tree.CJUMP.LE,new Tree.TEMP(s),
  new Tree.CONST(0),l5,l4),
new Tree.SEQ( new Tree.LABEL(l4),
```

---

Per ogni iterazione copiamo il valore di inizializzazione nella cella di memoria puntata da bb, quindi incrementiamo bb e decrementiamo s

---

```
new Tree.SEQ( new Tree.MOVE(new Tree.MEM(new Tree.TEMP(bb)),
  new Tree.TEMP(i)),
new Tree.SEQ( new Tree.MOVE(new Tree.TEMP(bb),
  new Tree.BINOP(Tree.BINOP.PLUS,new Tree.TEMP(bb),
  new Tree.CONST(k))),
new Tree.SEQ( new Tree.MOVE(new Tree.TEMP(s),
  new Tree.BINOP(Tree.BINOP.MINUS,new Tree.TEMP(s),
  new Tree.CONST(1))),
new Tree.SEQ( new Tree.JUMP(l3),
  new Tree.LABEL(l5) )))))))
new Tree.TEMP(b));
}
```

Si consideri per esempio il seguente programma Tiger:

```
let
  var i:int := 0
in
  while (i < 10) do i := i + 1
end
```

Esso viene compilato in

```

SEQ(
  MOVE(
    MEM(BINOP(PLUS,TEMP t0,CONST -4)),
    CONST 0),
  SEQ(
    LABEL L12,
    SEQ(
      CJUMP(LT,
        MEM(BINOP(PLUS,TEMP t0,CONST -4)),
        CONST 10,
        L13,L11),
      SEQ(
        LABEL L13,
        SEQ(
          MOVE(
            MEM(BINOP(PLUS,TEMP t0,CONST -4)),
            BINOP(PLUS,
              MEM(BINOP(PLUS,TEMP t0,CONST -4)),
              CONST 1)),
          SEQ(
            JUMP(NAME L12),
            LABEL L11))))))

```

L'espressione `MEM(BINOP(PLUS,TEMP t0,CONST -4))` restituisce il valore della variabile `i`. Inizialmente le si assegna 0. Poi si entra in un ciclo (che inizia alla etichetta `L12`) il quale controlla se `i` è minore di 10. In tal caso esegue il corpo del ciclo, altrimenti esce. Il corpo del ciclo incrementa `i` e salta all'inizio del ciclo.

## 5.7 Usiamo il generatore di codice intermedio

Il generatore di codice viene chiamato automaticamente da `Semant/Semant.java`. Dobbiamo solo preoccuparci di stampare il risultato della compilazione, cioè il corpo principale del programma e i frammenti di compilazione. Riscriviamo quindi la classe `Parse/Parse.java` come segue:

```
Parse/Parse_translate.java
```

```

package Parse;

public class Parse {
  public ErrorMessage.ErrorMessage errorMsg;
  public Absyn.Exp absyn;

  public Parse(String filename) {
    ...
    absyn = (Absyn.Exp)(symbol.value);
    System.out.println("Fine dell'analisi sintattica");

    if (absyn != null) {
      Semant.Semant s = new Semant.Semant(errorMsg);

```

---

Adesso il risultato della compilazione ci serve (si confronti con la sezione 4.7)

---

```

  Translate.Exp code = s.transProg(absyn);
  System.out.println("Fine dell'analisi semantica");

```

---

Se non ci sono stati errori, stampiamo il codice del corpo principale del programma e tutti i frammenti generati durante la compilazione

---

```

        if (!errorMsg.anyErrors) {
            System.out.println("\nmain:");
            s.codegen.print(code);
            System.out.println();
            s.codegen.printFrag();
            System.out.println("Fine della generazione del codice");
        }
    }
}
else System.out.println("Attributo semantico pari a null");
}
}

```

Il file `Parse/Main_translate.java` è sempre uguale a `Parse/Main_syntactical.java`, ma li teniamo distinti per eventuali future modifiche.

Aggiungiamo al `makefile` un nuovo target che compila il generatore di codice intermedio.

makefile

---

Per compilare il traduttore, lo copiamo come `Translate/Translate.java` e richiamiamo il compilatore Java. Il fatto di non averlo chiamato direttamente con tale nome ci permette di usare traduttori diversi per target diversi del `makefile`. Si consideri per esempio il caso del traduttore vuoto utilizzato per far funzionare la sola analisi semantica (Sezione 4.7)

---

```

translate_translate: Translate/Translate_translate.java \
    Translate/Translate.java
@echo "*** Compiling the translator ***"
cp Translate/Translate_translate.java Translate/Translate.java
javac ${JFLAGS} Translate/Translate.java
touch translate_translate

```

---

Questo è il target che genera la shell per il generatore di codice

---

```

translate: Parse/Grm.java Parse/Main_translate.java \
    Parse/Parse_translate.java translate_translate \
    Semant/Semant.class Parse/Main.class

```

---

Copiamo i files specifici della shell e compiliamola

---

```

@echo "*** Compiling the translate shell ***"
cp Parse/Parse_translate.java Parse/Parse.java
cp Parse/Main_translate.java Parse/Main.java
javac ${JFLAGS} Parse/Main.java
touch translate

```

---

Aggiorniamo l'elenco delle cose da cancellare ad ogni `make clean`

---

```

.PHONY: clean
clean:
    -rm lexical
    -rm syntactical
    -rm semantic
    -rm translate_semantic
    -rm translate_translate
    -rm Parse/*.class ErrorMessage/*.class Parse/Yylex.java
    -rm Parse/Grm.java

```

```
-rm Absyn/*.class Symbol/*.class
-rm Semant/*.class Translate/*.class
-rm Types/*.class Temp/*.class Translate/*.class
-rm Intel/*.class
-rm Frame/*.class rm FindEscape/*.class
-rm Tree/*.class
touch Parse/Main.class
```

Per compilare e provare il nostro generatore di codice intermedio basta dare i seguenti comandi:

```
> make translate
> java Parse.Main testcases/test1.tig
```